# MITOCW | Lecture 21 | MIT 6.832 Underactuated Robotics, Spring 2009

**RUSS TEDRAKE:** Welcome back. So today we get to finish our discussion on at least the first wave of value-based methods for trying to find optimal control policies, without a model. So we started last time talking about these model-free methods. And just to make sure we're all synced up here, so big picture is that we're trying to learn an optimal policy, approximate optimal policy, without a model, by just learning an approximate value function.

And the claim is that value functions are a good thing to learn for a couple of reasons. First of all, they should describe everything you need to know about the optimal control. Second of all, they're actually fairly compact. I'm going to say more about that in a minute. But if you think about it, probably a value function might actually be a simpler thing to represent than the policy, a smaller thing to represent it, because it's just a scalar value over all states.

And the third big motivation I tried to give last time was that these temporal difference methods which bootstrap based on previous experience, they're, like value iteration and dynamic programming, can be very efficient in terms of reusing the computation or reusing the samples that you've gotten by using estimates that you've already made with your value function to make better, fast estimates of your value function as you [INAUDIBLE]. These are all going to come up again. But that's just the high-level motivation for why we care about trying to learn value functions.

And then first thing we did-- it was really all we did last time. The first thing we had to achieve was just estimate value function for a fixed policy, which we called J pi, right? And we did it just from sample trajectories.

In discrete state and action, we called them s and a. Take a bunch of trajectories, and you would be able from those trajectories to try to back out J pi. And those trajectories are generated using policy pi.

So I actually tried to argue that that was useful even in the-- so if you just want-- if you have a robot out there that's already executing a policy, or a passive walker or something like this that doesn't have a policy, and you just want to see how well it's doing, estimate its stability by example, then you can actually-- this might be enough for you. You might just try to evaluate how well that policy is doing. We call that policy evaluation.

What we're actually interested it's not that. That's just the first step. What we care about now is, given if we can estimate the value for a stationary policy, can we now do something smarter and more involved, and try to estimate what the optimal value function. Or you might think of it as continuing to estimate the value function as we change pi towards the [INAUDIBLE] the optimal cost, the optimal policy.

We talked about a couple of ways to estimate the value function for a fixed pi, right? Even for function approximation, we did first Markov chains, and then we went to function approximation. And we have convergence results for linear function approximators.

And we went back and looked up [INAUDIBLE] it had a question about whether they used lambda in your update, if it always got to the same estimate of J. And I think the answer was, yes, it always gets-- the convergence proof has an error bound. And that error bound does depend on lambda, if you remember [INAUDIBLE] discussion. But if you said your learning rate gets smaller and smaller and you go, it should converge to the-- they should all converge to the same estimate of J pi.

So if you think about it, learning J pi shouldn't involve any new machinery, right? If I'm just experiencing cost, and I'm experiencing states, and I'm trying to learn a function of cost-to-go given states, that should just be able to do a least squares function. It's just a standard function approximation task. I could just do just a least squares function approximation, least squares estimation, and what we call the Monte-Carlo error. Just run a bunch of trials, figure out the estimates of what the cost-to-go was at every time, and then just do least squares estimation.

The machinery we developed last time was because it's actually a lot faster using bootstrapping algorithms. [INAUDIBLE] much faster than [INAUDIBLE]. Right. So we talked about the TD lambda algorithm, including for function approximation.

The only reason we had to develop any new machinery is because we wanted to be able to essentially do the least squares estimation, but we wanted to reuse our current estimate as we build up the estimate. And that's why it's not just a standard function approximation task we did all the [INAUDIBLE] something [INAUDIBLE] presented it.

OK. So that's the simple policy evaluation story. Now the question is, how do we use the ability to do policy evaluation to get towards a more optimal policy? So today, given the new policy evaluation, we want to improve the policy. And the idea of this-- the first idea you have to have in your head, very, very simple. And it's called policy iteration.

So given I start off with some initial guess for a policy, and I run it for a little while, I could do policy evaluation. So I'm converged on a nice estimate to get J pi 1. And now I'd like to take J pi 1, my estimate, and come up with a new pi 2. We've talked about how the value function infers a policy.

And if I repeat, and I do it properly, then if all goes well, I should find myself-- if it's always increasing in performance, and we can show that, then I should find myself eventually at the optimal policy and optimal value function, right? So we said TD lambda was a candidate for sitting there and evaluating policy, which I've talked about a couple of different ways to do policy evaluation.

So the question now is, how do we this, then? That's the first question. So given your policy, given your value function, how do you compute a new policy that's at least as good as your own policy but maybe better?

**AUDIENCE:** Maybe stochastic gradient descent?

**RUSS TEDRAKE:** Do something like stochastic gradient descent? You have to be careful with stochastic gradient. You have to make sure it's always going down, and things like that. It's a good idea. In fact, that's sort of-- actually, [INAUDIBLE]. We combine stochastic gradient descent and evaluation to do actor-critic [INAUDIBLE].

But there's a simpler sort of idea. I guess the thing it requires-- I didn't even think about this when I was making the notes. But I guess it requires an observation that-- so the optimal value function and the optimal policy have a property that the policy is going, taking the fastest descent down the value function. Your job is to go down the value function as fast as possible.

But if you're not optimal yet, I've got some random policy, and I figure out my value of executing that policy, that's actually not true yet. So what I need to say is, if you start giving your value function, you come up with a new policy which tries to be as aggressive as possible on this value function, which in our continuous sense, is going down the gradient of the value function as fast as possible. And that should be at least as good-- in the case of the optimal policy, it should be the same. It should return the optimal policy again.

But in the case where the value estimates from another, original policy gets you to do better. So the basic story-- that's the continuous gradient-- is you want to come up with a greedy policy that moves down, that does the best it can with this J pi. So pi 2, let's say, which is a function of s, should be, for instance, [INAUDIBLE] the discrete sense here, discrete state and action, minimize the expected value [INAUDIBLE] expected value first, by one-step error plus--

So I've got the cost that I incur here plus the long-term cost here. I want to pick the new min over a. The best thing I can do given that estimate of the value function. And that's going to give me a new policy, actually, pi 2, which is greedy with respect to this estimate of the value function. What does that look like to you guys?

**AUDIENCE:** [INAUDIBLE]

**RUSS TEDRAKE:** Yeah. OK. So value iteration, or dynamic programming, is exactly policy iteration in the case where you do a sweep through your entire state space every time, and then you update, sweep your entire state space, you do the update. Absolutely.

But it's a more general idea than just value iteration. You don't have to actually evaluate all s. You might call it asynchronous value-- [INAUDIBLE]?

**AUDIENCE:** Shouldn't that be argmin [INAUDIBLE]?

**RUSS TEDRAKE:** Oh, good. Thank you, yeah. This is argmin. Good catch. Yeah.

**AUDIENCE:** This is like g [INAUDIBLE].

**RUSS TEDRAKE:** Right. I always minimize this. So g is [? bad. ?] Well, I don't promise that I will never make a mistake with the signs, because I try to use reinforcement [INAUDIBLE] notation with costs, and I can sometimes get myself into trouble. I never write "arg." It's always g.

OK. So so this would be argmin. The min is the value estimated in the case of value iteration. But in general, you don't have to wait till you sweep the entire state space. You can just take a single trajectory through, update your value J. Or you take lots of trajectories through [INAUDIBLE] get an improved estimate for J, and then do this and get a new policy, right?

In this policy iteration, the original idea is you should really do this policy evaluation step until your estimate of J pi convergence, and then move on. But in fact, value iteration and other-- many algorithms show that you can-- it actually is still stable when you don't wait for it to converge.

But there's a problem with what I wrote here. I don't think there's a technical problem. But why is that not quite what we need for today's lecture? Yeah.

**AUDIENCE:** I just had a quick question. So if you're going to be gradient with respect to the value function that you evaluate, you can't do that with a value function if you have a model, right? So you need a--

**RUSS TEDRAKE:** That's actually exactly-- you're answering the question that I was asking. That's perfect. So from, as I said, model-free, model-free, model-free, but then I wrote down a model here. So how can I-- even in the steepest descent sort of continuous sense, this is absurd.

In the discrete sense, argmin over a is typically done with a search over all actions in the continuous state and action. I think it was finding the gradient down the slope. But right. Both of those require a model to actually do that policy [INAUDIBLE].

So the first question for today is, how do we come up with a gradient policy, basically, without any model? [INAUDIBLE] going to say it. [INAUDIBLE] know this, but that's the-- what do you think?

[INAUDIBLE] haven't read all the [INAUDIBLE] algorithms. What do you think? What's the-- how could I possibly come up with a new policy without having a model?

**AUDIENCE:** [INAUDIBLE] s n plus [INAUDIBLE] sample directly?

**RUSS TEDRAKE:** Good. You could sample. You can start to do some local search to come up with [INAUDIBLE]. Turns out-- I mean, I didn't actually ask the question in a way that anybody would have answered it in the way I wanted, so.

So it turns out if we changed the thing we store just a little bit, then it turns out to contribute to do model-free greedy policy. OK. So the way we do that is a Q function. We need to find a Q function. It's a lot like a value function. But now it's a function of state and action.

And we'll say this is still [INAUDIBLE] this way. OK. So what's a Q function? A Q function is the cost you should expect to take, to incur, given you're in a current state and you take a particular action. So it's a lot like a value function. But now you're actually learning a function over both state and actions.

So in any state, Q pi is the cost I should expect to incur given I take action a for one step and I follow a policy pi for the rest of the time. That make sense?

So I could have my acrobot controller or something like this. And in a current state, I've got a policy that mostly gets me up, but I'm learning more than just what that policy would do from this state. I'm learning what that policy would have done if I had for one step executed any random action on the function, for any random action. And then what would I do from the-- beginning I ran that controller for the rest of it.

Algebraically, it's going to make a lot of sense why we would store this. But it's actually interesting to think a little bit about what that Q function should look like. And if you have a Q function, you certainly could also get the value function, because you can look up for a given pi what action that policy would have taken. You can always pull out your current value function from Q.

But you can also-- [INAUDIBLE] simple relationship here in the [INAUDIBLE]. And for the optimal [INAUDIBLE], I should actually do that search over a. I almost wrote minus. That can be your job for the day, make sure I don't flip any signs.

OK. We're roboticists in this room. What does it mean to learn a Q function? What are the implications of learning a Q function?

Well, I guess I didn't say. So given the Q function pi [INAUDIBLE] having a Q function makes action collection easy. Pi 2 of s is now just a min over a Q pi s and a, where Q pi was [INAUDIBLE] with pi 1.

**AUDIENCE:** [INAUDIBLE]

**RUSS TEDRAKE:** It's argmin [INAUDIBLE]. But I was willing to-- the reason to learn a Q function in one case here is that it tells me about the other actions I could have taken. And if I want to now improve my policy, then I'll just look at my Q function. At every state I'm in, instead of taking the one at pi a, I'll go ahead and take the best one.

If pi 1 was optimal, that I would just get back the same policy. But if pi 1 wasn't optimal, then I'll get back something better, given my current estimate of Q. OK.

But what does it mean to run Q? And this is actually all you need to learn to do model-free value [INAUDIBLE] optimal policy. That's actually really big. So it's a little bit more to learn than learning a value function. And you're learning about your [INAUDIBLE].

If I had to learn J pi, how big is that? If I'm going to say I've got n dimensional states and m dimensional u-- I'll just think about these two new cases, even though [INAUDIBLE] this. If I have to learn J pi, how big is that? What's that function mapping for?

**AUDIENCE:** [INAUDIBLE] scalar learning.

**RUSS TEDRAKE:** Learning a scalar function over the state space to R1, just learning a scalar function. If I was learning a policy, how big would that be? If I was learning a stationary policy, it might be that.

So how bad is it to learn Q? What's Q?

**AUDIENCE:** [INAUDIBLE] asymptote [INAUDIBLE].

**RUSS TEDRAKE:** Let's keep it a deterministic policy for now.

**AUDIENCE:** [INAUDIBLE]

**RUSS TEDRAKE:** Yeah. Now I've suddenly got to learn something over-- sorry, [INAUDIBLE] here.

**AUDIENCE:** [INAUDIBLE]. Yeah, there.

**RUSS TEDRAKE:** OK. And for [INAUDIBLE], what would it be used to learn a modeled system? If I wanted to use this idea. What's that model?

[INTERPOSING VOICES]

**RUSS TEDRAKE:** Yeah. So f and then n plus m to Rn. So let's just think about how much you have to learn. So the easiness of learning this is not only related to the size. But it does matter.

So most of the time, as control guys, as robotics guys we would probably try to learn a model first, and then do model-based control. The last few days I've been saying, let's try to do some things without learning a model.

Here's one interesting reason why. It's actually-- learning a model is sort of a tall order. It's a lot to learn, right? You've got to learn from every possible state and action what's my x dot [INAUDIBLE]. This is only learning from every possible state and action. What's the expected cost-to-go [INAUDIBLE]? [INAUDIBLE] a scalar. So this is learning one algorithm for all m.

And the beautiful thing about optimal control, with this sort of additive cost functions and everything like that, the beautiful thing is that this is all you need to know to make optimal decisions. You don't need to know your model. That model is extra information.

All you need to know to make optimal decisions, given these additive cost functions [INAUDIBLE] is given [INAUDIBLE] a state and then a given action, how much do I expect to incur cost [INAUDIBLE]? It's a beautiful thing.

So if we make it stochastic, it gets even sort of-- learning a stochastic model, if your dynamics are variable and that's important, you want to do stochastic optimal control. Learning a stochastic model is probably even harder than that. Maybe I have to learn the mean of x dot plus the covariance matrix of x dot or something like this. When I use a stochastic model, it would be even more expensive.

Q, in the Q sense-- I left it off in the first pass just to keep it clean, but Q is just going to be the expected value around this. So Q is always going to be a scalar, even in the stochastic optimal control sense.

So maybe this is the biggest point of optimal control, honestly-- optimal control related to learning-- is that if you're willing to do these additive expected value optimization problems, which I think you've seen lots of interesting problems that fall into that category, then all you need to know to make decisions is to be able to-- the value function, the Q function here. The expected value of future penalties. And for everything else, [INAUDIBLE]. Important point.

Now, just to soften it a little bit, in practice, you might not get away with only that. If you have to somehow build an observer to do state estimation or to estimate Q, and you've got-- there might be other reasons floating around in your robot that might require you to learn this. But in a pure sense, that's really what you need to know.

**AUDIENCE:** Hey, Russ?

**RUSS TEDRAKE:** Yeah.

**AUDIENCE:** You put x and u. Shouldn't that be s and--

**RUSS TEDRAKE:** Right. We could have-- I could have said n is the number of states.

**AUDIENCE:** I just meant, should it be s and a?

**RUSS TEDRAKE:** Right. I would have-- I wrote the dimension of x, and I called it Rn,m. So that's what I meant. If you want to make an analogy back here, then it would actually be just the number of elements in s and a. But I wanted to sort of be a roboticist [INAUDIBLE] for a little bit.

**AUDIENCE:** OK.

**RUSS TEDRAKE:** This is just the computer scientist that did this [INAUDIBLE]. But it does make this easier, so I still [INAUDIBLE]. So I meant to do that. So that [INAUDIBLE].

OK. So now, how do we learn Q? I told you how to learn J. Q looks pretty close to J. How do I learn Q?

I told you about temporal different learning, probably wouldn't have wasted your time talking about temporal difference learning if it wasn't also relevant for what we needed to do these model-free value methods. So let's just see that temporal difference learning also works for learning these functions. OK? That's just some [INAUDIBLE].

Let's do just a simple case first, where I'm just doing-- remember, TD0 was just bootstrapping. It wasn't carrying around long-term rewards. It was just saying [INAUDIBLE] one step, and then I'm going to use my value estimate for the rest of the time as my new update.

And I'll go ahead, since we're-- we talked about last time how a function approximator [INAUDIBLE] reduce it to the Markov chain case, let's just do it like-- let's say [INAUDIBLE] of s is alpha i phi i, linear function approximators. Or we could, in fact, reduce alpha t phi s, a.

OK. Then the TD lambda update it going to be-- TD0 update is just going to be alpha plus gamma call that hat just to be careful here-- pi s transpose.

These really are supposed to be s n and a n. I get a lot of [INAUDIBLE] for my sloppy [INAUDIBLE]. OK. And Q pi-- or the gradient here in the linear function approximator case, is just phi s, a. So if you look back in your notes, that's exactly what we had before, where this used to be J.

We're going to use is our new update-- we're going to say that our new estimate for J is basically the one-step cost plus the long-term look-ahead. a n plus 1 in the case of doing an on-policy-- if I'm just trying to do policy evaluation, it's going to be pi s n plus 1. We'll use that one-step prediction minus my current prediction and try to make that go to 0.

And in order to do it in a function approximator sense, that means multiplying that error, the temporal difference error, by the gradient. And that was something like gradient descent on your temporal difference policy. But not exactly, because it's this whole recursive dependence thing.

People get why-- do people get that it's not quite gradient descent but kind of this? This looks a lot like what I would get if I was trying to do gradient descent [INAUDIBLE], right? But only in the case of TD1 was actually gradient descent. But normally if I have a y minus f of x, I'm trying to do the gradient with respect to this, I've got to minimize this. And I'll get something-- if I take the gradient with respect to alpha, I get the error alpha x [INAUDIBLE].

**AUDIENCE:** [INAUDIBLE]

**RUSS TEDRAKE:** Because what we got here, this is our error. If we assume that this is just my desired and this is my actual, then this is gradient descent. But it's not quite that, because this depends on an alpha-- these all depend on alpha. So by virtue of having this one in the alpha, it's not exactly gradient descent algorithm.

But it still works. People proved that it works. Is that OK? And actually, in the case where TD is one, these things actually go through it and cancel each other out with whatever is a gradient descent algorithm [INAUDIBLE].

But I want you to see, this is my error I'm trying to make Q and my current state and action look like my one-step cost plus Q of my next state and action. And I would do that by multiplying my error by my gradient in a gradient descent kind of idea.

OK. You can still do TD lambda if you like also. Q functions And the big idea there was to use an eligibility trace, which in the function approximator case, was gamma lambda ei n plus [INAUDIBLE]. And then my update is the same thing-- alpha-- because this is my big temporal difference error. And instead of multiplying by the gradient [INAUDIBLE] this eligibility trace.

And magically through an algebraic trick, remembering the gradient computes the bootstrapping case when lambda is 0, and the Monte Carlo case when lambda is 1, and something in between when lambda is [INAUDIBLE]. OK. So you'd still do temporal difference there.

Big point number two-- big idea number one is we have to use Q functions to do action selection. Big point number two is off-policy policy evaluation. Once we start using Q, you could do this trick that I mentioned first thing we're doing value methods. And that is to execute policy pi 1 but learn Q pi 2 [INAUDIBLE]. Can you see how we do that?

By virtue of having this extra dimension, we know we're learning-- bless you-- not only what happens when I take policy pi from state s. I'm learning what happens when I take any action in state s. That gives me a lot more power. Because for instance, when I'm making my temporal difference error, I don't need to necessarily use my one-step prediction as the current policy. I can just look up what would policy 2 [INAUDIBLE].

Because I'm storing every state-action pair, it's more to learn, more work. But it means I can say, I'd like my new Q pi 2 to be the one-step policy I got from taking a plus the long-term cost of taking policy pi 2. And then all the same equations play out, and you get-- you get an estimate for policy 2.

**AUDIENCE:** Does it count--

**RUSS TEDRAKE:** Yeah?

**AUDIENCE:** Does it count more than the first step of the policy 2 and then take your cost-to-go of policy 1? Or does it somehow--

**RUSS TEDRAKE:** So I can't switch-- we'll talk about whether you can switch halfway through. But once I commit to learning Q pi 2, then actually this whole thing is built up of experience of executing policy 2, even though I've only generated sample paths for policy 1. So it's a completely consistent estimator of Q pi 2, right?

If I halfway through decided I wanted to start evaluating pi 3, then I'm going to have to wait for those cancel out, or we play some tricks to do that. But it actually recursively builds up in the estimator of pi 2.

**AUDIENCE:** Can I ask a question?

**RUSS TEDRAKE:** Of course.

**AUDIENCE:** [INAUDIBLE] have that [INAUDIBLE] function like this, we can substitute y--

**RUSS TEDRAKE:** You're talking about this?

**AUDIENCE:** Yes. You can substitute y by [? g or ?] gamma, and then execute the--

**RUSS TEDRAKE:** Yeah.

**AUDIENCE:** And then take the derivative?

**RUSS TEDRAKE:** Yes.

**AUDIENCE:** Why [INAUDIBLE]?

**RUSS TEDRAKE:** So why isn't it true gradient descent? That's exactly what I proposed to do. But the only problem is, this isn't what we have. What we actually have is this, which means that this is not the true gradient [INAUDIBLE] term for partial y partial from over here.

**AUDIENCE:** That's what I'm suggesting. So instead of y alpha, we can actually [INAUDIBLE] g plus gamma-- an approximation of y. So this g plus gamma Q is [INAUDIBLE] approximation for y, right?

**RUSS TEDRAKE:** I'm trying to perfectly make the analogy that this looks like that, and this looks like that.

**AUDIENCE:** Right. But when we're taking the derivative from that function, we assume that y is constant.

**RUSS TEDRAKE:** Yes.

**AUDIENCE:** And then solve this.

**RUSS TEDRAKE:** Yes.

**AUDIENCE:** We can actually assume that y is dependent on alpha and the derivative of that term with respect to alpha as well, and then solve it.

**RUSS TEDRAKE:** Yes. So you could do that. So you're saying why don't we actually have a different update which has the gradient [INAUDIBLE]? OK, good.

So in the case of TD0-- TD1, you actually do have that. And I think that's true. I worked this out a number of years ago. But I think it's true that if you start including that, if you look at the sum over a chain, for this standard update with TD0, for instance, that those terms, this term now will actually cancel itself out on this term here, for instance.

It doesn't work. It doesn't work nicely. It would give you-- it gives you back the Monte Carlo error. It doesn't do temporal difference. It doesn't do the bootstrapping. So basically, you start including that, then you do get a least squares algorithm, of course. But it's effectively doing Monte Carlo. You have to sort of ignore that do to temporal difference learning.

You're actually saying, I'm going to believe this estimate in order to do that. OK? Temporal difference, if you actually want to prove any of these things, I have one example of it in the note. I think that I put in TD1 is gradient descent in the notes, just so you see an example.

A story-- rule of the game in temporal difference learning, derivations, and proofs, is you start expanding these sums, and terms from time n and terms from time n plus 1 cancel each other out in a gradient way. And you're left with something much more compact. That's why [? everybody ?] calls it an algebraic trick, why these things work.

But because these are not random samples drawn one at a time, they're actually directly related to each other, that's why it makes it more complicated. OK. So we said off-policy evaluation says, execute policy pi 1 to get pi 1 generates s n a n trajectories.

But you're going to do the update alpha plus-- I'm going to just write quickly here-- g s, a plus gamma Q pi-- this is going to be estimator Q pi 2-- s n plus 1 pi 2. What would pi 2 have done in kind of s n plus 1? In general, we'll multiply it by [INAUDIBLE].

That's a really, really nice trick. Let's learn about policy 1-- or policy 2 while we execute policy 1. OK. So what policy 2 should we learn about? And again, these are-- I'm asking the questions in bizarre ways, and there's a specific answer. But [INAUDIBLE] ask that question.

Our ultimate goal is not to learn about some arbitrary pi 2. I want to learn about the optimal policy. I don't have the optimal policy. But I have an estimate of it. So actually, a perfectly reasonable update to do, and the way you might describe it is, let's execute-- I'm putting it in quotes, because it's not entirely accurate, but it's the right idea-- execute policy 1 but learn about the optimal policy.

And how would we do that? Well-- this is now my shorthand Q star here. Estimate of Q star is s n plus 1-- I should have--

It makes total sense. Might as well, as I'm learning, always try to learn about policy which is optimal with respect to my current estimate J [? hat. ?] And this algorithm is called Q-learning. OK. It's the crown jewel of the value-based methods [INAUDIBLE].

I would say it was the gold standard until probably about [? '90-- ?] something like that. When people started to do policy gradient stuff more often. Even probably halfway through the '90s, people were still mostly [INAUDIBLE] papers about Q-learning. and there was a movement in policy gradient.

**AUDIENCE:** So is your current estimate Q star not based on pi 1?

**RUSS TEDRAKE:** It is based on data from pi 1. But if I always make my update, making it this update, then it really is learning about pi 2.

**AUDIENCE:** Isn't pi 2 what you're computing with this update?

**RUSS TEDRAKE:** Good. There's a couple of ways that I can do this. So in the policy-- in the simple policy iteration, we use [INAUDIBLE] evaluate for a long time, and then you make an, update, you evaluate for a long time, and you make an update.

**AUDIENCE:** This is dynamically--

**RUSS TEDRAKE:** This is always sort of updating, right? [INAUDIBLE]. And you can prove that it's still a sound algorithm despite [INAUDIBLE]. This is always sort of updating its policy as it goes. Compared to this, which is more of the-- learn about pi 2 for a while, stop, [INAUDIBLE] pi 3 for a while, stop, this is trying to go straight through pi.

OK, good. So what is it-- what's required for a Q-learning algorithm to converge? So even for this algorithm to converge, in order for pi 1 to really teach me everything there is to know about pi 2, there's some important feature, which is that pi 1 and pi 2 had better pick the same actions with some old probability.

So off-policy works. Let's just even think about-- I'll even [INAUDIBLE] first in the discrete state and discrete actions and the Markov-- MDP formulations. Off-policy works if pi 1 takes in general all state-action pairs with some small probability. If pi 2 took action [INAUDIBLE] state 1 and pi 1 never did, there's no way I'm going to learn really what pi 2 is all about. [INAUDIBLE] show you those two [INAUDIBLE].

OK. So how do you do that? If you just-- if you're thinking about greedy policies on a robot, and you've got your current estimate of the value, and you do the most aggressive action on the acrobot, I'll tell you what's going to happen. You're going to visit the states near the bottom, and you start learning a lot. And you're never going to visit the states up at the top when you're learning.

So how are you going to get around that on the acrobot? And the acrobot is tough, actually. But the idea is, you'd better add some randomness so that you explore more and more state and actions. And the hope is that if you add enough for a long enough time, you're going to learn better and better policies, you're going to find your way up to the top.

So the acrobot is actually almost as hard as it gets with these things, where you really have to find your way into this region to learn about the region. In fact, my beef with reinforcement learning community is that they learn only to swing up to some threshold. They never actually [INAUDIBLE] to the top.

If you look, there's lots of papers, countless papers, written about reinforcement learning, Q-learning for the acrobot and things like this, and they never actually solve the problem. They just try to [INAUDIBLE] at the top. But they don't do it. They just get up this high. Because it is sort of a tough case.

So how do you do this? So you get-- like I said, in order to start exploring that space, you'd better add some randomness. So one of the standard approaches is to use epsilon greedy algorithms.

So I said, let's make pi 2 exactly the minimizing thing. That's true. But if you execute that, you're probably going to [INAUDIBLE] much better to execute a policy pi s, where the-- let's say the policy I care about, I'm going to execute with probability-- sort of, you flip a coin, you pull a random number between 0 and 1. If it's greater than epsilon, then go ahead and execute policy you're trying to learn about, but execute some random action otherwise.

OK. So every time I-- every dt I'm going to flip a coin, keep it-- well, not a coin. A hundred-sided coin, a 100 to-- 0 to 1, a continuous thing. If it comes out less than epsilon, I'm going to do a random action. Just forget about my current policy, pick a random action. It's a uniform distribution over actions. Otherwise, I'll take this, the action from my policy.

And by virtue of having a soft policy learning thing is I can still learn about pi 2, even if I'm taking this pi epsilon. But I have the advantage of exploring all the state-actions. Good. I'm missing a page.

**AUDIENCE:** Is that the most randomness you can produce since they'll [INAUDIBLE] converge?

**RUSS TEDRAKE:** There's a couple of different candidates. The softmax is another one that people use a lot. And a lot of-- I mean, in the off-policy sense, it's actually quite robust. So a lot of people talk about using behavioral policy, which is just sort of something to try-- it's designed to explore the state space. Actually, my candidate for behavioral policy is something like RRT. We should really try to do something that gets me into all areas of state space, for instance.

And then maybe that's a good way to design, to sample these state-action pairs. And all the while, I try to learn about pi 2. So it is robust in that sense.

When I say it works here, I have to be a little careful. This is only for the MDP case that it's really guaranteed to work. There's more recent work doing off-policy and function approximators. And you can do that.

I don't want to bury you guys with random detail. But you can do off-policy with linear function approximators safely, using an important [INAUDIBLE] when you're dealing [INAUDIBLE]. And that's work by Doina Precup. The basic idea is you have to-- if your policy is changing, like these things, it's changing over time, you'd better weight your updates based on the relative--

**AUDIENCE:** [INAUDIBLE] necessarily. But that's--

**RUSS TEDRAKE:** But what you're learning about is the state-- the probability of picking this action for one step and then executing pi 2. And that's still [INAUDIBLE] even pi 2 would never take that action.

**AUDIENCE:** Oh, yeah. Because it's-- OK.

**RUSS TEDRAKE:** So I think it's good. The thing that has to happen is that pi 2 has to be well-defined for every possible state.

**AUDIENCE:** OK. So keeping the Q pi 2 is take a certain [INAUDIBLE] take certain action, then [INAUDIBLE] pi [INAUDIBLE].

**RUSS TEDRAKE:** Yes.

**AUDIENCE:** OK. Sorry, I lost that [INAUDIBLE].

**RUSS TEDRAKE:** OK, good. Sorry. Thank you for clarifying it. Yeah, so cool. So I think that still works. OK. So this is good. So let me tell you where you are so far.

We've now switched from doing temporal difference learning on value functions and temporal difference learning on Q functions. And a major thing we got out of that was that we can do this off-policy learning.

You put it all together. [INAUDIBLE] back into my policy iteration diagram, and what we have, we've defined the policy evaluation, that's the TD lambda. We defined our update, which could be this in the general sense. This one is-- if I used pi 1 again, If I really did on-policy, if I used pi 1 everywhere while I'm executing pi 1, then this would be called SARSA, [INAUDIBLE] sort of on-policy Q-learning, on-policy updating.

And Q-learning is this, where you use the middle gradient. And what we know, what people have proven, the algorithms were in use for years and years and years before it was actually proven, even in the tabular case, where you have finite state and actions. But now we know that this thing is guaranteed to converge to the optimal policy, that policy iteration, even if it's updated at every step, is going to converge to the optimal policy and the optimal Q function, given that all state-action pairs are [INAUDIBLE] in the tabular case.

If we go to function approximation, if you just do policy evaluation but not update, then we have an example where this is actually in '02 or something like that. It'd be '01 or '02, 2001. We finally proved that off-policy with linear function approximation would converge when the policy is not changing-- no control.

So the thing I need to give you before we consider this a complete story here is, can we do off-policy learning? Can we do our policy improvement, update stably with function approximation? And the algorithm that we have for that is our least squares policy iteration.

Do you remember least squares temporal difference learning? Sort of the idea was that if we look at the stationary update-- maybe I should write it down again. [INAUDIBLE] find it. If I look at the stationary update, if I were to run an entire batch of-- I mean, the big idea is when you're doing the least squares, is that we're going to try to reuse old data. We're not going to just make a single update, spit it out, throw9t away.

We're going to remember a bunch of old state-action pairs, trying to make a least squares update with just the same thing as [INAUDIBLE]. In the Monte Carlo sense, it's easy. It's just function approximation, where with this TD term floating around it's harder. So we had to come up least squares temporal difference learning.

And in the LSTD case, the story was we could build up a matrix using something that looked like phi of s gamma phi transpose-- so it's ik. Let me just write ik in here-- ik plus 1 minus phi of ik. times my parameter vector. And b-- some of these terms was e, which were phi ik times our reward times--

And if I did this least squares solution-- or I could invert that carefully with SVD or something like that-- then what I get out is the-- it jumps immediately to steady-state solution TD lambda.

So this is essentially the big piece of the TD lambda update broken into the part that depends on alpha, the part that doesn't depend on alpha. I could write my batch TD lambda update as alpha equals alpha plus gamma a alpha plus b. And I could solve that at steady-state [INAUDIBLE].

All right. So least squares policy, least squares temporal difference learning, is about reusing lots of trajectories to make a single update that was going to jump right to the case where the temporal difference learning we've gotten through. We just replayed it a bunch of times. Now the question is, the policy is going to be moving while we're doing this. How can we do this sort of least squares method to do the policy iteration up here?

Again, the trick is pretty simple. We've just got to learn the Q function instead, [INAUDIBLE] biggest trick. So in order to do-- control not just evaluate a single policy but actually try to find the optimal policy, first thing we have to do is figure out how to do LSTD on a Q function. And it turns out it's no-- yeah, what's up? [INAUDIBLE].

It turns out if you keep along, [INAUDIBLE] exactly the same form as we did in least squares temporal difference learning, but now we do everything with functions of s and t. [INAUDIBLE] transpose on it.

Now I do the-- you said form of the [INAUDIBLE] too much. I just want you to know the big idea here. Then I do gamma [INAUDIBLE] a inverse b, this whole time we're representing our Q function. Q hat s, a is now a linear combination of nonlinear basis functions on s and a.

**AUDIENCE:** Shouldn't that be transpose [INAUDIBLE]?

**RUSS TEDRAKE:** I put-- I tried to put a transpose with my poorly-- throughout everything. So you're saying this one shouldn't be transpose?

**AUDIENCE:** [INAUDIBLE] should be a [INAUDIBLE]?

**RUSS TEDRAKE:** Yeah. Good. So I'm going to-- but this one, I wrote this whole update as the transpose of the other-- of what I just wrote over there.

**AUDIENCE:** [INAUDIBLE] write alpha? [INAUDIBLE] If it was a plus b some stuff?

**RUSS TEDRAKE:** Yeah. And there's an alpha, sorry. Thank you. Well, actually, it's-- the alpha is really not there. Yeah, I should have written it here. It's a times alpha. That's what it makes the update on. So we get [INAUDIBLE] alpha. So that is actually [INAUDIBLE].

This is the one I did [INAUDIBLE]. OK, good. So it turns out you can learn a Q function just like you can learn a value function, with storing up these matrices, which are what TD learning would have done in a batch sense, and then just taking a one-step shot to get directly to the solution for temporal difference learning for Q function.

And again, I put in here s prime. And I left this a little bit ambiguous. So I can evaluate any policy by just putting in that policy in here and doing the replay. And it turns out if I now do this in a policy iteration sense, LSPI-- Least Squares Policy Iteration-- basically, you start off with an initial guess, we do LSTDQ [INAUDIBLE] to get Q pi 1. And then you repeat, yeah?

Then this thing, that's enough to get you to-- it converges. Now, be careful about how it converges. It converges with some error bound to pi star Q star. The error bound depends on a couple of parameters. So technically, it could be close to your solution and oscillate, or something like that. But it's a pretty strong convergence result for this for policy improvement with approximate value function.

In a pure sense, it is-- you should run this for a while until you get a good estimate for LSTD and you get your new Q pi. But by virtue of using Q functions, when you do switch to your new policy, pi 2, let's say, you don't have to throw away all your old data. You just take your old tapes and actually regenerate a and b as if you had played off those old tapes with-- as if you had seen the old tapes executing the new policy. And you can reuse all your old data and make an efficient update to get Q pi [INAUDIBLE].

Least squares policy iteration. Pretty simple. OK. I know that was a little dry and a little bit-- and a lot. But let's make sure we know how we got where we got.

So there's another route besides pure policy search to do model-free learning. All you have to do is take a bunch of trajectories, learn a value function for those trajectories. You don't even actually have to take the perfect-- your best controller yet. You could take some RRT controller, something that's going to explore the space and try to learn about your value function.

Learn Q pi through these LSTD algorithms, you could do a pretty efficient update for doing Q pi. You can improve efficiently by just looking over the min over Q, and pretty quickly iterate to an optimal policy and optimal value function, only storing-- the only thing you have to store in that whole process is the Q function.

And in the LS case, the LSTD case, you remember the tape-- the history of tapes, just you use them more efficiently. Yeah?

**AUDIENCE:** So could you have used this on the flapper that John showed? Or what's the--

**RUSS TEDRAKE:** Good. Very good question. That's an excellent question. So in fact, the last day of class, what we're going to do is going to-- the last day I present in class, we're going to-- I'm going to go through a couple of sort of case studies and different problems that people have had success and tell you why we picked the algorithm we picked, things like that.

So why didn't we do this on a flapper? The simplest reason is that we don't know the state space. It's infinite dimensional in general. So that would have been a big thing to represent a Q function for.

It doesn't mean-- it doesn't make it invalid. We could have learned, we could have tried to approximate the state space with even a handful of features, learned a very approximate Q function, and done something like actor-critic like we're going to do next time. But I think in cases where you don't know the state space, or the state space is a very, very large, and you can write a simple controller, then it makes more sense to parameterize the policy.

It really goes down to that game, that accounting game, in some ways, of how many dimensions things are. But in a fluids case, you could have a pretty simple policy from sensors to actions which we could twiddle. We couldn't have an efficient value function.

Now, there are other cases where the opposite is true. The opposite is true, where you have a small state space, let's say, but the resulting policies would require a lot of features to parameterize. But I think in general, the strength of these algorithms is that they are efficient with reusing data. The weakness is that-- well, the weakness a few years ago would have been that they big blow up the time. But algorithms have gotten better as we [INAUDIBLE] we have some convergence guarantees.

Not the general [INAUDIBLE]. I never told you that it converged if you have a nonlinear function approximator. We'd love to have that result [INAUDIBLE]. We won't have it for a while. But in linear function approximator sense, we have both. But there's a lot of success stories.

These are the kind of algorithms that we're used to play backgammon. There are examples of them working on things like the [INAUDIBLE]. But in the domains that I care about most in my lab, we tend to do more policy gradient sort of things.